ISO/IEC JTC 1/SC 22/WG14 N2429

September 23, 2019 v 1

Function failure annotation

Niall Douglas
ned Productions Ltd, Ireland

Jens Gustedt
INRIA & ICube, Université de Strasbourg, France

We have been seeing an evolution in proposals for the best syntax for describing how to mark how a function fails, such that compilers and linkers can much better optimise function calls than at present.

These papers were spurred by [WG21 P0709] *Zero-overhead deterministic exceptions*, which proposes deterministic exception handling for `C++`, whose implementation could be very compatible with `C` code.

P0709 resulted in [WG21 P1095] *Zero overhead deterministic failure – A unified mechanism for* `C` *and* `C++` which proposed a `C` syntax and exception-like mechanism for supporting `C++` deterministic exceptions.

That was responded to by [WG14 N2361] *Out-of-band bit for exceptional return and errno replacement* which proposed an even wider reform, whereby the most popular idioms for function failure in `C` code would each gain direct calling convention support in the compiler.

This paper synthesises all of the proposals above into common proposal, and does not duplicate some of the detail in the above papers for brevity and clarity. It hews closely to what [WG14 N2361] proposes, but with different syntax, and in addition it shows how the corresponding `C++` features can be constructed on top of it, without loosing call compatibility with `C`.

Contents

1. INTRODUCTION

[WG14 N2361] *Out-of-band bit for exceptional return and errno replacement* describes different function failure mechanisms in the standard C library:

— **mech_ub**
Invalid input leads to undefined behavior. E.g many character classification macros such as **isalpha** are undefined if the input is not either a valid character or **EOF**.
— **mech_sentinel**
These return a failure exclusively by returning a sentinel value (e.g. -1, NaN, **EOF** or a null pointer), or a range of sentinel values (e.g. < 0). Many of the C standard library functions work like this, the simplest example being **getc** which returns a value **EOF** unequal to any character value in case of failure.
— **mech_sentinel+errno**
These indicate a failure via returning a sentinel value as above, and additionally store the cause of the failure in a thread-local state such as **errno**, or **fegetexceptflag()**. In a variant of this, **mech_errno+sentinel**, say, the sentinel value (such as **INT_MAX**) only indicates a potential failure, and **errno** has to be checked to learn the failure state.
— **mech_errno**
These return a failure exclusively by writing to a thread-local state e.g. **errno** not being modified means function was successful (i.e. to check for failure, the caller must set **errno** to zero before the call). Any function return value may be a valid value, or garbage.

In this context the picture is complicated by POSIX (ISO 9945) because in many cases it changes the category from the one foreseen by C (ISO-9899). *E.g* C has the **malloc** function in category **mech_sentinel**, whereas POSIX sets error codes into **errno**, and thus changes the category to **mech_sentinel+errno**.

Failure modes **mech_ub**, **mech_sentinel+errno** and **mech_errno** are difficult to handle and to optimize, and would nowadays probably have been designed differently. *E.g* modifying thread local data, which appears as-if modifying a global variable to the C abstract machine in terms of side effects, prevents function declarations from being marked as pure by the developer and generally complicates code generation. This prevents much optimisation which would otherwise be possible, particularly pessimising small functions like the C math functions which in standard C configuration must spend more time in function ceremony than implementing the mathematical operation itself.

N2361 emphasises a lot on improving this situation of C legacy interfaces to provide them with three alternative calling mechanisms on one side, and by providing three alternative return specifications for failure on the other. Each alternative return specification for each function would be an alternative implementation edition of that function. Each implementation edition is proposed to use a fixed offset from a linker symbol, but other implementations are possible, and we will not focus on this aspect here.

— **N2361's oob_direct**
Caller expects legacy behaviour i.e. set thread-local state or similar or have even undefined behavior in case of failure. This is the default calling mechanism that is chosen when the function is called directly.
— **N2361's oob_ignore**
Caller ignores failure, so any failure-related code in the callee is wasted CPU cycles, and that part of that code could be optimized out.
— **N2361's oob_plain**
Caller *requires* that the function must never fail, invalid input will still produce a return value.
— **N2361's oob_capture**
An additional local channel for propagating a failure bit can be captured by the caller. By

receiving such a failure bit, the return value of the function can be interpreted differently and *e.g* be used to transport additional failure information. Other path of failure communication (**errno**, `fesetexcept()`, ...) are cut off and are guaranteed not to be changed by this call mechanism.

For legacy interfaces that use failure mechanisms **mech_sentinel+errno** or **mech_errno**, **oob_capture** means that a caller has to be able to handle the fact that the thread-local state is not being modified, and that the error value is returned directly by hijacking the function return value, instead.

As previously mentioned, N2361 also provides alternative return mechanisms for the functions such that they may specify failure. The most important of these for the following will be **oob_return** which simply indicates 'failure' of the execution and that provides the possibility to return additional failure information through the specified return value. Two other failure modes are provided, **oob_return_never** and **oob_return_error**. These are mainly thought to provide means to amend legacy interfaces that follow one of the failure conventions above.

Under N2361, each function declaration would be annotated with attributes to indicate that it can be called with any of the additional calling mechanisms – if it is not annotated at all it is a legacy function and calling it with the new modes would not be defined. These annotations enable compile-time rather than link-time diagnostics. Also they improve optimisation of calls, because now the caller may know if a function could set **errno** eventually or not.

The caller must always explicitly opt-in to calling a non-legacy edition by wrapping the function call with markup to indicate that an optimised calling mechanism is requested. If code tries to call a function compiled by a legacy compiler other than directly (**oob_direct**), where the function declaration is incorrectly attributed, a link error would always result.

The additional calling mechanisms are designed such they can be used with identifiers of such annotated functions or by using conventional function pointers to these functions.

1.1. Differences between WG14 N2361 and WG21 P1095

[WG21 P1095] *Zero overhead deterministic failure – A unified mechanism for* `C` *and* `C++` spent much of its time describing how existing **errno**-based `C` code could be mechanistically transformed by the compiler into code which no longer used **errno** under the bonnet, but which otherwise did not need to be rewritten apart from opting-in a function definition. This was intended to aid existing code to be recompiled and automatically 'just work' in the majority of cases, with diagnostics where code would need to be modified. WG14 did not much care for this approach, as it is felt that `C` compilers ought to not rewrite the programmer's code to such an extent, even if mechanistically.

N2361 simply requires that the programmer must rewrite their code, particularly in the implementation of a function, but also at every function call site if one does not want legacy behaviour.

The broader sweep of N2361 of tackling the entire standard `C` library at once is a useful innovation. One can get onboard the proposal that function implementations must be reimplemented to support more efficient calling conventions, as that would be a once-off investment of effort by the library implementer.

What is unhelpful in N2361 is the requirement for code calling standard library functions to have to wrap every call site with markup to specifically opt into the more efficient calling convention.

— Firstly, that is tedious, but it is also visually fussy, ruining the clarity and succinctness of `C` prose.

— Secondly, one specifically want existing code to automatically be more efficient, where the new behaviour has identical visible side effects to the current behaviour, when recompiled with new compilers and new standard libraries.
— Thirdly, how N2361 would interoperate with C++ deterministic exceptions is not obvious, and a slightly different formulation of N2361 would solve that problem.

2. PROPOSED DESIGN

Returning to the sample C standard library function from WG21 P1095R0 which we shall permute to demonstrate the proposal at work:

```
int myabs(int x) {
  if(x == INT_MIN) {
    errno = ERANGE;
    return INT_MIN;
  }
  return (x < 0) ? -x : x;
}
```

In the following we will show some 'editions' of this function that will highlight several aspects of our proposal. Whether or not these 'editions' may be specified directly (as the following example suggests) or if they will only be deduced automatically from the supplementary information that is provided to the compiler, will only be decided in the future depending to the feedback from the committees and the wider public. They are mainly used here to communicate and clarify the optimisation potential of the approach.

These functions are annotated with a particular 'qualifier' such as `:ignore_errno`, that is voluntarily chosen to be syntactically different from any C or C++ syntax. Also, for illustration a gcc attribute `[[gnu::const]]` is used to indicate that a given function is considered to be referentially transparent, but other optimization attributes could equally be used.

For the header file, if we wish to declare to the public that `myabs()` supports more efficient failure handling, we would declare our `myabs()` implementations to external code as follows:

```
#define oob_errno_combine_decl(T) struct result_with_errno_##T \
{                                                              \
  T return_value;                                              \
  int errcode;                                                 \
}
#define oob_errno_combine(T) struct result_with_errno_##T

// Input is not checked edition.
[[gnu::const]] extern int myabs:ignore_errno(int x);

// Product type return edition.
[[gnu::const]] extern oob_errno_combine(int) myabs:struct_errno(int x);

// Invalid input is UB edition.
[[gnu::const]] extern int myabs:signal_error(int x);

// errno setting edition.
// Linkage is "myabs" as well as "myabs:raise_errno", as :raise_errno is
// default edition (see below)
extern int myabs:raise_errno(int x);
```

The following implementations then follow:

```
#include <errno.h>
#include <limits.h>

// Input is not checked edition
```

```
[[gnu::const]] int myabs:ignore_errno(int x) {
  // maybe undefined if x is INT_MIN
  return (x < 0) ? -x : x;
}

// Return what errno would be set to put into a product type.
[[gnu::const]] oob_errno_combine(int) myabs:struct_errno(int x) {
  // expected to be false, could be annotated and optimized
  if(x == INT_MIN) {
    return (oob_errno_combine(int)){INT_MIN, ERANGE};  // sentinel value + errno
  }
  return (oob_errno_combine(int)){myabs:ignore_errno(x), 0};
}

// Invalid inputs cause UB due to precondition violation.
[[gnu::const]] int myabs:signal_error(int x) {
  oob_errno_combine(int) ret = myabs:struct_errno(x);
  // expected to be false, could be annotated and optimized
  if(ret.errcode != 0) {
    abort();
  }
  return ret.return_value;
}

// Set errno if failed. Not pure. Default edition.
int myabs:raise_errno(int x) {
  oob_errno_combine(int) ret = myabs:struct_errno(x);
  // expected to be false, could be annotated and optimized
  if(ret.errcode != 0) {
    errno = ret.errcode;
  }
  return ret.return_value;
}
```

It should be stressed that this will ever only be an opt-in facility for the library implementer. Here, it gives us the opportunity to talk about the different editions of the function. Nobody is suggesting that any of the above is required for any given function implementation.

Note also that whilst a library implementer *could* implement all of the above manually by hand, later in this paper we propose convenience markup to avoid writing so much tedious boilerplate.

2.1. `errno`-related function failure editions

The `ignore_*`, `struct_*`, `union_*`, `signal_*` and `raise_*` edition qualifiers have a pattern to them, though the exact semantics are hardcoded to each edition qualifier. For `*_errno`:

— `ignore_errno`
  No setting nor checking of **`errno`** will be performed by the caller. Therefore, any code related to the use of **`errno`** to return failure ought to be elided in this edition.
— `struct_errno`
  This edition will return a sentinel value and what **`errno`** would be set to in a product type (i.e. a structure), or the successful value with the error code set to zero. **`errno`**'s value on input is never read.
— `union_errno`
  This edition will return either the successful value, or what **`errno`** would be set to in a sum type (i.e. a union), along with an efficient discriminant to indicate which is in force. In order to preserve the type system perfectly in function pointers, rather than returning

a **union**, such functions return a symbolic type `oob_union` so the compiler knows that an efficient discriminant such as the CPU's carry flag, or spare CPU register, was used.

As none of the `C` standard library functions ever return either a value or **errno**, none would implement a `union_errno` edition. This would not prevent other functions in user libraries doing so, however.

— `signal_errno`

Invalid inputs, or invalid input state, will cause the function to never return e.g. a signal may be raised instead, or the program may crash. This means that if the function returns, it was successful. This differs from `ignore_errno` because implementations will probably still detect failure, but is similar in that callers will never check for failure.

If the reader is familiar with `C++`'s `noexcept`, this is the exact same facility: no `C++` exception throws ever emit from a `noexcept` function, because `std::terminate()` would be called instead.

— `raise_errno` **[default]**

This edition sets **errno** if failure occurs. For the family of `*_errno` editions, `raise_errno` is defined as the **default**. That means that the `raise_errno` edition of the function is emitted into linkage as the unqualified edition i.e. this is the edition of the function which unrecompiled code links to, and if you take the address of the unadorned function, this is the function pointer type you get.

Note that by defining any `*_errno` edition of a function, you guarantee that that function never changes its behaviour based on *reads* of **errno**, and that the other editions have the same caller-visible side effects as the default edition, except for side effects upon **errno**.

There are many more proposed hardcoded `ignore_*`, `struct_*`, `union_*`, `signal_*` and `raise_*` edition qualifiers for other than **errno**, which we will cover later.

2.2. When to choose which edition

Before we get onto those however, the next part is how compilers ought to interpret the availability of multiple editions of functions. Such automatic optimisation opportunities had not been developed previously in N2361. We show such opportunities for **errno**-setting functions with `:struct_errno` editions available. Consider a sequence of code like the following where a loop of an **errno**-setting function accumulates failure:

```
int out[length], in[length];
errno = 0;  // reset errno to zero
for(size_t n = 0; n < length; n++) {
  out[n] = myabs(in[n]);
}
// If any one of the above myabs() functions failed ...
if(0 != errno) {
  abort();
}
```

An optimiser can spot that the setting of real **errno** can be hoisted outside of the loop, thus yielding as-if code:

```
int out[length], in[length];
int errcode = 0;  // NOT errno
for(size_t n = 0; n < length; n++) {
  oob_errno_combine(int) ret = myabs:struct_errno(in[n]);
  if(ret.errcode != 0) {
    out[n] = ret.return_value;
  }
  errcode |= ret.errcode;
}
if(0 != errcode) {
```

```
  abort();  // never returns
}
errno = 0;  // must be zero henceforth
```

Not setting a thread local **errno** state every loop iteration is much more efficient, plus myabs:struct_errno() is const pure, and thus calls of it can be more aggressively elided and/or parallelised via SIMD. The optimisation is safe, because the mere existence of *_errno editions for a function guarantee that replacing the default edition with a non-default edition does not have unknown side effects.

Similarly, consider a sequence of code like the following:

```
int out[length], in[length];
int saved_errno = errno;
for(size_t n = 0; n < length; n++) {
  out[n] = myabs(in[n]);
}
// errno changes are explicitly thrown away
errno = saved_errno;
```

Here, an optimiser can spot that setting **errno** has no visible side effects upon the caller, thus yielding as-if code:

```
int out[length], in[length];
for(size_t n = 0; n < length; n++) {
  out[n] = myabs:ignore_errno(in[n]);
}
```

If however the programmer did not save and restore **errno** and simply does not check it at all:

```
int out[length], in[length];
for(size_t n = 0; n < length; n++) {
  out[n] = myabs(in[n]);
}
// errno is not checked
```

and the compiler has been configured with some option like -funchecked-**errno**-functions -always-succeed, then the optimiser might generate:

```
int out[length], in[length];
for(size_t n = 0; n < length; n++) {
  out[n] = myabs:signal_errno(in[n]);  // failure not possible
}
```

One can easily see the optimisation opportunities if the compiler can hard-assume that an unchecked **errno**-setting function call will never set **errno** for a given translation unit:[1] it can optimise out failure handling, remove branches, improve CPU pipelining and vectorize.

2.3. Function pointers

If one were to choose an implementation where each function implementation edition has separate linkage, then each edition of a function would be an ordinary C function. It would be callable directly by specifying its edition, the address can be taken, and so on:

```
int (*ignore_myabs)(int) = &myabs:ignore_errno;

oob_errno_combine(int) (*struct_myabs)(int) = &myabs:struct_errno;
```

[1] It would probably be unwise to enable such a compiler setting for an entire program.

```
int (*signal_myabs)(int) = &myabs:signal_error;

int (*raise_myabs)(int) = &myabs:raise_errno;

int (*default_myabs)(int) = &myabs;

assert(raise_myabs == default_myabs);  // true
```

As one can see, the function pointer type system would be unmodified from the existing system.

What happens if somebody supplies an ignore edition to a function pointer whose caller expects an **errno** setting edition? This would be possible without casting as the pointer types are compatible. The answer is misoperation – don't do that.

2.4. Math function edition qualifiers

Of particular concern to this proposal is to improve the performance of the C math functions in standards conforming C code. WG21 is especially acutely keen to fix this, as standard C++ is impossible on GPUs due to the use of thread-local data, which GPUs do not have.

The following math function editions are such they add to the previous *_errno by also considering the floating point exception flags and environment, as they are defined in the <fenv.h> header. The following qualifiers are proposed:

— ignore_feexcept
  Assume that the caller immediately called **feholdexcept()** and saved **errno** just prior to calling this function, and will call **fesetenv()** and restore **errno** immediately after this function returns. The edition can therefore hard-assume that any floating-point failure will be ignored, including that **errno** will not be checked by the caller.

— struct_feexcept
  A structure compatible with the following layout would be returned:

```
{
  T return_value;  // The return value type e.g. float, double, long double,
      etc
  int feexceptcode;
}
```

  This edition will return either the successful value in return_value and 0 in feeexceptcode, or return a sentinel value in return_value and what **feraiseexcept()** would have been called with to raise the floating-point exception. **errno**'s value on input is never read, and **errno**'s value is never written.

— union_feexcept
  Unavailable as is unneeded in the C standard library.

— signal_feexcept
  Floating-point failure is handled by raising **SIGFPE**, and thus the current thread of execution must be aborted. The caller can assume that failure will never occur.

— raise_feexcept **[default]**
  This edition calls **feraiseexcept()** and/or sets **errno** if failure occurs as per the current rules for **math_errhandling**.

3. CONVENIENCE SYNTAX

Generating and maintaining different 'editions' of the same function is tedious and error prone; providing only the opportunity for handwritten optimisations as presented above would severely restrict the acceptability of this technique. Therefore, we propose convenience

syntax that will probably the more commonly used interfaces, if not the only ones that survive the scrutiny of the standardisation process.

3.1. Function declaration attributes

The different versions of `myabs` can basically be seen as mechanical rewriting of the same code. This may be due to the simplicity of the base function in question, or it could be a general property of the error convention using **`errno`** and the floating exception flags. In fact, a first inspection of a concrete implementation of the `C` library (musl libc) has shown that the error handling is concentrated in some easily identifiable error paths, and that an annotation of these can be done mostly automatically.

So, in order to save on spelling out function editions by hand, it is proposed that function attributes will trigger the automatic generation of the editions by the compiler, similar to N2361's proposed syntax. The earlier `myabs()` function could have the compiler autogenerate the `*_errno` editions simply by placing an `[[oob_return*]]` attribute before its declaration, and using a family of `oob_*()` features to save on writing much tedious boilerplate:

```
[[oob_return_errno(nondefault: gnu::const)]] int myabs(int x) {
  if(x == INT_MIN) {
    oob_return_errno(ERANGE, INT_MIN);
  }
  return (x < 0) ? -x : x;
}
```

Here, we assume a new feature **`oob_return_errno`**`(...)` (extending N2361, see below) which applies the attributes specified by `...` to a subset of the editions, depending on the rule specified. This allows one to announce that:

— the function is referentially transparent, provided no error condition occurs,
— in case of such an error condition only the default version would use **`errno`** to transmit the failure state,
— the function implementation solely uses the **`oob_return_errno`** feature to indicate failure, and,
— that the non-default editions receive the attribute `[[gnu::const]]`.

In the header file, the declaration also becomes similarly succinct:

```
[[oob_return_errno(nondefault: gnu::const)]] int myabs(int x);
```

With such simple annotations, different editions as proposed in the previous section can be automatically generated *and* automatically used. In the terminology of N2361 the automatically rewritten optimisation of the first loop example of the previous section can be as simple as the following:

```
int out[length], in[length];
int errcode = 0;  // NOT errno
for(size_t n = 0; n < length; n++) {
  bool flag;
  out[n] = oob_capture(&flag, myabs, in[n]);
  errcode |= flag;
}
if(0 != errcode) {
  abort();  // never returns
}
// errno has never been touched
```

Such a rewrite then would behave as-if it were the original code. On the other hand, this rewritten code can be optimized much more efficiently because the loop does not contain any conditional and because the access to the thread local state **errno** can completely be avoided.

In view of the different editions that we have introduced above, we propose to differentiate the different error return features from N2361. In particular, following the above terminology the **oob_return_error** convention would better be spelled **oob_return_feexcept**, and we add a convention **oob_return_errno** that guarantees that default failure paths exclusively use **errno**.

3.2. WG21 P0709 Deterministic C++ Exceptions

As this is a WG14 focused paper, this section will be extremely brief, but it is worth demonstrating how C code could safely and usefully call many more C++ functions in the future than at present.

Assuming that [WG21 P1028] *SG14* `status_code` *and standard error object for P0709 Zero-overhead deterministic exceptions* becomes the chosen `std::error` for lightweight exceptions, implementing C support for calling C++ functions which throw lightweight exceptions is straightforward.

Under the [WG21 P1095] formulation of P0709, deterministic exception throwing functions can locally throw a to-`std::error` convertible custom type more appropriate to a local use case. Such custom local type throws must be some custom `std::status_code<DomainType>`, which can implicitly decay to `std::error` upon demand. Under P1095, status codes know how to throw themselves as a non-deterministic exception upon request.

As `std::error` is also a `std::status_code<erased<T>>`, deterministic exception throwing functions would always throw some `status_code`. Hence we can hard code C++ function edition semantics around those of `status_code`:

— `ignore_status(erased<E>/DomainType)`
  The status code will never be checked by calling code, and thus any code related to checking whether to return a status code can be skipped in the edition.
— `struct_status(erased<E>/DomainType)`
  This edition will return a sentinel value and what the status code would be in a product type (i.e. a structure), or the successful value with the status code set to empty.
  There is currently no proposed use case for this edition in the core C++ language. However, there are a number of places in the C++ standard library where the ability to return a value with informational status would be extremely useful.
— `union_status(erased<E>/DomainType)`
  This function implementation edition is selected by the C++ language when a C++ `throws` function calls another C++ `throws` function i.e. a deterministic exceptions enabled function is calling another deterministic exception enabled function.
  This edition will return either the successful value, or what the status would be in a sum type (i.e. a union), along with an efficient discriminant to indicate which is in force.
  This edition is **always** generated by C++ `throws(E)` or `throws` functions i.e. deterministic exception throwing C++ functions. As mentioned above, pointers to functions with this edition would return a `oob_union` type, which indicates an ABI-efficient means of returning a discriminant (but see subsection below on function pointers).
— `signal_status(erased<E>/DomainType)`
  This function implementation edition is selected when a `noexcept` function calls a C++ `throws` function, and does not trap nor check the deterministic exception thrown.
  This edition never returns failure i.e. is a `noexcept` C++ function. If it experiences failure, it calls `std::terminate()`, as all `noexcept` C++ functions do.

Note that this is an optimisation: the compiler is effectively hoisting the potential call to `std::terminate()` out of the `noexcept` function, and into the source of the uncaught exception throw. This would mean that destructors within the `noexcept` function, or any destructors between the leafmost throwing function and the `noexcept` function, would not fire before `std::terminate()` is called. However, it also means that codegen for `noexcept` functions dramatically improves, and it also solves a big problem with value-based exceptions in that the cause of why `std::terminate()` was invoked gets lost much easier than with type-based exceptions. By invoking `std::terminate()` at the very leafmost function, instead of at the `noexcept` base of the call tree, it localises the cause.

— `raise_status(DomainType)`

This function implementation edition is selected when an exception type throwing function calls a `throws` function i.e. when a legacy exception throwing function calls a deterministic exception throwing function.

Upon failure, this edition raises a non-deterministic type-based exception throw by calling the status code's `.throw_exception()` facility to cause the status code to be converted into a legacy exception throw.

Depending on the `C` implementation, throwing type-based exceptions through `C` code may be unwise. Portable `C` code should not call this function implementation edition.

3.2.1. `C` compatibility

`C++` status codes with a `C`-compatible payload are guaranteed to be `C`-compatible, and, from the perspective of `C`, they have a layout of:

```
struct status_code_##DomainPayloadType
{
  void *domain;               // pointer to status_code_domain instance
      describing T
  DomainPayloadType payload;  // An intptr_t for std::error
};
```

So, `C++` functions look just like `C` functions with editions returning a status code as their failure code, and for `struct_status(erased<E>/DomainType)`, `union_status(erased<E>/DomainType)` and `signal_status(erased<E>/DomainType)`, `C` code can directly call `C++` functions, and understand their failures, without intermediate wrappers to prevent non-deterministic type-based exception throws entering `C` code.

3.2.2. `C++` function pointers

Something important to note is that in this proposal, deterministic exception value throwing functions do not have a compatible function pointer type with type-based exception throwing functions i.e.

```
template<class T>
T make() throws;

template<class T>
static T (*make_function_ptr)() = &make<T>;  // FAILS, incompatible pointer types
```

Instead one must either spell out the function pointer type, or annotate the function pointer type with `throws`:

```
// C++ style
template<class T>
static T (*make_function_ptr1)() throws = &make<T>;                 // works

// C style
template<class T>
```

```
static oob_union(T, std::error) (*make_function_ptr2)() = &make<T>;  // works
```

This solves a concern raised in the Cologne EWG discussion of P0709 about mixing pointer types of deterministic exception throwing functions with other kinds of exception throwing functions.

But it also solves another desire: the C++ style function pointer is *auto-propagating* i.e. when calling that function pointer, C++ will automatically insert a check for failure after the function call returns, and will propagate any failure up the call tree. In contrast, the C style function pointer does not propagate – like in C, calling it gets you a returned value of oob_union type, and it is on the caller to inspect that value and do something with it.

This proposed facility raises some very interesting possibilities for header files which contains function declarations which auto-propagate failure when used from C++, and return oob_union when used from C code. This facility is not explored further in this paper.

4. ACKNOWLEDGEMENTS

Our thanks to the numerous people who have contributed feedback on lightweight exceptions. Your numbers are so numerous now that listing you all would be distracting from this paper's goal to be a short, combined proposal for consideration by the standards committees. Nevertheless, thanks are due to each and every one of you.

5. REFERENCES

References

Herb Sutter,
Zero-overhead deterministic exceptions
https://wg21.link/P0709

Douglas, Niall
SG14 status_code *and standard error object for P0709 Zero-overhead deterministic exceptions*
https://wg21.link/P1028

Douglas, Niall
Zero overhead deterministic failure – A unified mechanism for C *and* C++
https://wg21.link/P1095

Gustedt, Jens
Out-of-band bit for exceptional return and errno replacement
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2361.pdf